

Contents

1 Setting	1	7 String	10
1.1 vimrc	1	7.1 KMP	10
2 Math	1	7.2 Aho-Corasick	10
2.1 Basic Arithmetic	1	7.3 Suffix Array with LCP	11
2.2 Sieve Methods : Prime, Divisor, Euler phi	2	7.4 Suffix Tree	11
2.3 Primality Test	2	7.5 Manacher's Algorithm	11
2.4 Chinese Remainder Theorem	3	8 Miscellaneous	11
2.5 Burnside's Lemma	3	8.1 Fast I/O	11
2.6 Kirchoff's Theorem	3	8.2 Magic Numbers	11
2.7 Fast Fourier Transform	3		
2.8 Matrix Operations	3		
2.9 Gaussian Elimination	4		
2.10 Simplex Algorithm	4		
3 Data Structure	5	1 Setting	
3.1 Order statistic tree	5	1.1 vimrc	
3.2 Fenwick Tree	5	set ts=4 sts=4 sw=4	
3.3 Segment Tree with Lazy Propagation	5	set ai si nu	
3.4 Persistent Segment Tree	6	2 Math	
3.5 Link/Cut Tree	6	2.1 Basic Arithmetic	
4 DP	6	typedef long long ll;	
4.1 Convex Hull Optimization	6	typedef unsigned long long ull;	
4.2 Divide & Conquer Optimization	6		
4.3 Knuth Optimization	6		
5 Graph	6	// calculate ceil(a/b)	
5.1 SCC (Tarjan)	6	// a , b <= (2^63)-1 (does not cover -2^63)	
5.2 SCC (Kosaraju)	6	ll ceildiv(ll a, ll b) {	
5.3 2-SAT	6	if (b < 0) return ceildiv(-a, -b);	
5.4 BCC, Cut vertex, Bridge	7	if (a < 0) return (-a) / b;	
5.5 Lowest Common Ancestor	7	return ((ull)a + (ull)b - 1ull) / b;	
5.6 Heavy-Light Decomposition	7	}	
5.7 Bipartite Matching (Hopcroft-Karp)	8		
5.8 Maximum Flow (Dinic)	8	// calculate floor(a/b)	
5.9 Min-cost Maximum Flow	9	// a , b <= (2^63)-1 (does not cover -2^63)	
6 Geometry	9	ll floordiv(ll a, ll b) {	
6.1 Basic Operations	9	if (b < 0) return floordiv(-a, -b);	
6.2 Compare angles	9	if (a >= 0) return a / b;	
6.3 Convex Hull	9	return -(ll)((ull)(-a) + b - 1) / b;	
6.4 Polygon Cut	10	}	
		// calculate a*b % m	
		// x86-64 only	
		ll large_mod_mul(ll a, ll b, ll m)	
		{	

```

    return ll((__int128)a*(__int128)b%m);
}

// calculate a*b % m
// |m| < 2^62, x86 available
// O(logb)
ll large_mod_mul(ll a, ll b, ll m)
{
    a %= m; b %= m; ll r = 0, v = a;
    while (b) {
        if (b&1) r = (r + v) % m;
        b >>= 1;
        v = (v << 1) % m;
    }
    return r;
}

// calculate n^k % m
ll modpow(ll n, ll k, ll m) {
    ll ret = 1;
    n %= m;
    while (k) {
        if (k & 1) ret = large_mod_mul(ret, n, m);
        n = large_mod_mul(n, n, m);
        k /= 2;
    }
    return ret;
}

// calculate gcd(a, b)
ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

// find a pair (c, d) s.t. ac + bd = gcd(a, b)
pair<ll, ll> extended_gcd(ll a, ll b) {
    if (b == 0) return { 1, 0 };
    auto t = extended_gcd(b, a % b);
    return { t.second, t.first - t.second * (a / b) };
}

// find x in [0,m) s.t. ax === gcd(a, m) (mod m)
ll modinverse(ll a, ll m) {
    return (extended_gcd(a, m).first % m + m) % m;
}

// calculate modular inverse for 1 ~ n
void calc_range_modinv(int n, int mod, int ret[]) {
    ret[1] = 1;
    for (int i = 2; i <= n; ++i)
        ret[i] = (ll)(mod - mod/i) * ret[mod%i] % mod;
}

```

2.2 Sieve Methods : Prime, Divisor, Euler phi

```

// find prime numbers in 1 ~ n
// ret[x] = false -> x is prime
// O(n*loglogn)
void sieve(int n, bool ret[]) {
    for (int i = 2; i * i <= n; ++i)
        if (!ret[i])
            for (int j = i * i; j <= n; j += i)
                ret[j] = true;
}

// calculate number of divisors for 1 ~ n
// when you need to calculate sum, change += 1 to += i
// O(n*logn)
void num_of_divisors(int n, int ret[]) {
    for (int i = 1; i <= n; ++i)
        for (int j = i; j <= n; j += i)
            ret[j] += 1;
}

// calculate euler totient function for 1 ~ n
// phi(n) = number of x s.t. 0 < x < n && gcd(n, x) = 1
// O(n*loglogn)
void euler_phi(int n, int ret[]) {
    for (int i = 1; i <= n; ++i) ret[i] = i;
    for (int i = 2; i <= n; ++i)
        if (ret[i] == i)
            for (int j = i; j <= n; j += i)
                ret[j] -= ret[j] / i;
}

```

2.3 Primality Test

```

bool test_witness(ull a, ull n, ull s) {
    if (a >= n) a %= n;
    if (a <= 1) return true;
    ull d = n >> s;
    ull x = modpow(a, d, n);
    if (x == 1 || x == n-1) return true;
    while (s-- > 1) {
        x = large_mod_mul(x, x, n);
        x = x * x % n;
        if (x == 1) return false;
        if (x == n-1) return true;
    }
    return false;
}

// test whether n is prime
// based on miller-rabin test
// O(logn*logn)
bool is_prime(ull n) {
    if (n == 2) return true;
    if (n < 2 || n % 2 == 0) return false;

    ull d = n >> 1, s = 1;

```

```

    for(; (d&1) == 0; s++) d >>= 1;

#define T(a) test_witness(a##ull, n, s)
    if (n < 4759123141ull) return T(2) && T(7) && T(61);
    return T(2) && T(325) && T(9375) && T(28178)
        && T(450775) && T(9780504) && T(1795265022);
#undef T
}

```

2.4 Chinese Remainder Theorem

```

// find x s.t.  x === a[0] (mod n[0])
//              === a[1] (mod n[1])
//              ...
// assumption: gcd(n[i], n[j]) = 1
ll chinese_remainder(ll* a, ll* n, int size) {
    if (size == 1) return *a;
    ll tmp = modinverse(n[0], n[1]);
    ll tmp2 = (tmp * (a[1] - a[0]) % n[1] + n[1]) % n[1];
    ll ora = a[1];
    ll tgcd = gcd(n[0], n[1]);
    a[1] = a[0] + n[0] / tgcd * tmp2;
    n[1] *= n[0] / tgcd;
    ll ret = chinese_remainder(a + 1, n + 1, size - 1);
    n[1] /= n[0] / tgcd;
    a[1] = ora;
    return ret;
}

```

2.5 Burnside's Lemma

경우의 수를 세는데, 특정 transform operation(회전, 반사, ..)해서 같은 경우들은 하나로 친다. 전체 경우의 수는?

- 각 operation마다 이 operation을 했을 때 변하지 않는 경우의 수를 센다 (단, “아무것도 하지 않는다”라는 operation도 있어야 함!)

- 전체 경우의 수를 더한 후, operation의 수로 나눈다. (답이 맞다면 항상 나누어 떨어져야 한다)

2.6 Kirchoff's Theorem

그래프의 스패닝 트리의 개수를 구하는 정리.

무향 그래프의 Laplacian matrix L 를 만든다. 이것은 (정점의 차수 대각 행렬) - (인접행렬)이다. L 에서 행과 열을 하나씩 제거한 것을 L' 라 하자. 어느 행/열이든 관계 없다. 그래프의 스패닝 트리의 개수는 $\det(L')$ 이다.

2.7 Fast Fourier Transform

```

void fft(int sign, int n, double *real, double *imag) {
    double theta = sign * 2 * pi / n;
    for (int m = n; m >= 2; m >>= 1, theta *= 2) {
        double wr = 1, wi = 0, c = cos(theta), s = sin(theta);
        for (int i = 0, mh = m >> 1; i < mh; ++i) {
            for (int j = i; j < n; j += m) {
                int k = j + mh;
                double xr = real[j] - real[k], xi = imag[j] - imag[k];
                real[j] += real[k], imag[j] += imag[k];
                real[k] = wr * xr - wi * xi, imag[k] = wr * xi + wi * xr;
            }
            double _wr = wr * c - wi * s, _wi = wr * s + wi * c;
            wr = _wr, wi = _wi;
        }
    }
    for (int i = 1, j = 0; i < n; ++i) {
        for (int k = n >> 1; k > (j ^= k); k >>= 1);
        if (j < i) swap(real[i], real[j]), swap(imag[i], imag[j]);
    }
}

// Compute Poly(a)*Poly(b), write to r; Indexed from 0
// O(n*logn)
int mult(int *a, int n, int *b, int m, int *r) {
    const int maxn = 100;
    static double ra[maxn], rb[maxn], ia[maxn], ib[maxn];
    int fn = 1;
    while (fn < n + m) fn <= 1; // n + m: interested length
    for (int i = 0; i < n; ++i) ra[i] = a[i], ia[i] = 0;
    for (int i = n; i < fn; ++i) ra[i] = ia[i] = 0;
    for (int i = 0; i < m; ++i) rb[i] = b[i], ib[i] = 0;
    for (int i = m; i < fn; ++i) rb[i] = ib[i] = 0;
    fft(1, fn, ra, ia);
    fft(1, fn, rb, ib);
    for (int i = 0; i < fn; ++i) {
        double real = ra[i] * rb[i] - ia[i] * ib[i];
        double imag = ra[i] * ib[i] + rb[i] * ia[i];
        ra[i] = real, ia[i] = imag;
    }
    fft(-1, fn, ra, ia);
    for (int i = 0; i < fn; ++i) r[i] = (int)floor(ra[i] / fn + 0.5);
    return fn;
}

```

2.8 Matrix Operations

```

const int MATSZ = 100;

inline bool is_zero(double a) { return fabs(a) < 1e-9; }

// out = A^(-1), returns det(A)
// A becomes invalid after call this
// O(n^3)
double inverse_and_det(int n, double A[][MATSZ], double out[][MATSZ]) {
    double det = 1;
    for (int i = 0; i < n; i++) {

```

```

    for (int j = 0; j < n; j++) out[i][j] = 0;
    out[i][i] = 1;
}
for (int i = 0; i < n; i++) {
    if (is_zero(A[i][i])) {
        double maxv = 0;
        int maxid = -1;
        for (int j = i + 1; j < n; j++) {
            auto cur = fabs(A[j][i]);
            if (maxv < cur) {
                maxv = cur;
                maxid = j;
            }
        }
        if (maxid == -1 || is_zero(A[maxid][i])) return 0;
        for (int k = 0; k < n; k++) {
            A[i][k] += A[maxid][k];
            out[i][k] += out[maxid][k];
        }
    }
    det *= A[i][i];
    double coeff = 1.0 / A[i][i];
    for (int j = 0; j < n; j++) A[i][j] *= coeff;
    for (int j = 0; j < n; j++) out[i][j] *= coeff;
    for (int j = 0; j < n; j++) if (j != i) {
        double mp = A[j][i];
        for (int k = 0; k < n; k++) A[j][k] -= A[i][k] * mp;
        for (int k = 0; k < n; k++) out[j][k] -= out[i][k] * mp;
    }
}
return det;
}

```

2.9 Gaussian Elimination

2.10 Simplex Algorithm

```

// Two-phase simplex algorithm for solving linear programs of the form
//      maximize    c^T x
//      subject to   Ax <= b
//                  x >= 0
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal solution will be stored
// OUTPUT: value of the optimal solution (infinity if unbounded
//         above, nan if infeasible)
// To use this code, create an LPSolver object with A, b, and c as
// arguments. Then, call Solve(x).
typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;
const double EPS = 1e-9;

```

```

struct LPSolver {
    int m, n;
    VI B, N;
    VVD D;

    LPSolver(const VVD& A, const VD& b, const VD& c) :
        m(b.size()), n(c.size()), N(n + 1), B(m), D(m + 2, VD(n + 2)) {
        for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) D[i][j] = A[i][j];
        for (int i = 0; i < m; i++) { B[i] = n + i; D[i][n] = -1; D[i][n + 1] = b[i]; }
        for (int j = 0; j < n; j++) { N[j] = j; D[m][j] = -c[j]; }
        N[n] = -1; D[m + 1][n] = 1;
    }

    void pivot(int r, int s) {
        double inv = 1.0 / D[r][s];
        for (int i = 0; i < m + 2; i++) if (i != r)
            for (int j = 0; j < n + 2; j++) if (j != s)
                D[i][j] -= D[r][j] * D[i][s] * inv;
        for (int j = 0; j < n + 2; j++) if (j != s) D[r][j] *= inv;
        for (int i = 0; i < m + 2; i++) if (i != r) D[i][s] *= -inv;
        D[r][s] = inv;
        swap(B[r], N[s]);
    }

    bool simplex(int phase) {
        int x = phase == 1 ? m + 1 : m;
        while (true) {
            int s = -1;
            for (int j = 0; j <= n; j++) {
                if (phase == 2 && N[j] == -1) continue;
                if (s == -1 || D[x][j] < D[x][s] || D[x][j] == D[x][s] && N[j] < N[s]) s = j;
            }
            if (D[x][s] > -EPS) return true;
            int r = -1;
            for (int i = 0; i < m; i++) {
                if (D[i][s] < EPS) continue;
                if (r == -1 || D[i][n + 1] / D[i][s] < D[r][n + 1] / D[r][s] ||
                    (D[i][n + 1] / D[i][s]) == (D[r][n + 1] / D[r][s]) && B[i] < B[r]) r = i;
            }
            if (r == -1) return false;
            pivot(r, s);
        }
    }

    double solve(VD& x) {
        int r = 0;
        for (int i = 1; i < m; i++) if (D[i][n + 1] < D[r][n + 1]) r = i;
        if (D[r][n + 1] < -EPS) {
            pivot(r, n);
            if (!simplex(1) || D[m + 1][n + 1] < -EPS)

```

```

        return -numeric_limits<double>::infinity();
    for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
            if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] &&
                N[j] < N[s]) s = j;
        pivot(i, s);
    }
}
if (!simplex(2))
    return numeric_limits<double>::infinity();
x = VD(n);
for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n + 1];
return D[m][n + 1];
}
};

```

3 Data Structure

3.1 Order statistic tree

```

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
#include <ext/pb_ds/detail/standard_policies.hpp>
#include <functional>
#include <iostream>
using namespace __gnu_pbds;
using namespace std;

// tree<key_type, value_type(set if null), comparator, ...>
using ordered_set = tree<int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update>;

int main()
{
    ordered_set X;
    for (int i = 1; i < 10; i += 2) X.insert(i); // 1 3 5 7 9
    cout << boolalpha;
    cout << *X.find_by_order(2) << endl; // 5
    cout << *X.find_by_order(4) << endl; // 9
    cout << (X.end() == X.find_by_order(5)) << endl; // true

    cout << X.order_of_key(-1) << endl; // 0
    cout << X.order_of_key(1) << endl; // 0
    cout << X.order_of_key(4) << endl; // 2
    X.erase(3);
    cout << X.order_of_key(4) << endl; // 1
    for (int t : X) printf("%d ", t); // 1 5 7 9
}

```

3.2 Fenwick Tree

```

const int TSIZE = 100000;
int tree[TSIZE + 1];

// Returns the sum from index 1 to p, inclusive
int query(int p) {
    int ret = 0;
    for (; p > 0; p -= p & -p) ret += tree[p];
    return ret;
}

// Adds val to element with index pos
void add(int p, int val) {
    for (; p <= TSIZE; p += p & -p) tree[p] += val;
}

```

3.3 Segment Tree with Lazy Propagation

```

// example implementation of sum tree
const int TSIZE = 131072; // always 2^k form && n <= TSIZE
int segtree[TSIZE * 2], prop[TSIZE * 2];
void seg_init(int nod, int l, int r) {
    if (l == r) segtree[nod] = dat[l];
    else {
        int m = (l + r) >> 1;
        seg_init(nod << 1, l, m);
        seg_init(nod << 1 | 1, m + 1, r);
        segtree[nod] = segtree[nod << 1] + segtree[nod << 1 | 1];
    }
}
void seg_relax(int nod, int l, int r) {
    if (prop[nod] == 0) return;
    if (l < r) {
        int m = (l + r) >> 1;
        segtree[nod << 1] += (m - l + 1) * prop[nod];
        prop[nod << 1] += prop[nod];
        segtree[nod << 1 | 1] += (r - m) * prop[nod];
        prop[nod << 1 | 1] += prop[nod];
    }
    prop[nod] = 0;
}
int seg_query(int nod, int l, int r, int s, int e) {
    if (r < s || e < l) return 0;
    if (s <= l && r <= e) return segtree[nod];
    seg_relax(nod, l, r);
    int m = (l + r) >> 1;
    return seg_query(nod << 1, l, m, s, e) + seg_query(nod << 1 | 1, m + 1, r,
        s, e);
}
void seg_update(int nod, int l, int r, int s, int e, int val) {
    if (r < s || e < l) return;
    if (s <= l && r <= e) {
        segtree[nod] += (r - l + 1) * val;
        prop[nod] += val;
        return;
    }
}

```

```

    seg_relax(nod, l, r);
    int m = (l + r) >> 1;
    seg_update(nod << 1, l, m, s, e, val);
    seg_update(nod << 1 | 1, m + 1, r, s, e, val);
    segtree[nod] = segtree[nod << 1] + segtree[nod << 1 | 1];
}
// usage:
// seg_update(1, 0, n - 1, qs, qe, val);
// seg_query(1, 0, n - 1, qs, qe);

```

3.4 Persistent Segment Tree

3.5 Link/Cut Tree

4 DP

4.1 Convex Hull Optimization

$O(n^2) \rightarrow O(n \log n)$

조건 1) DP 점화식 꼴

$$D[i] = \min_{j < i} (D[j] + b[j] * a[i])$$

조건 2) $b[j] \leq b[j + 1]$

특수조건) $a[i] \leq a[i + 1]$ 도 만족하는 경우, 마지막 쿼리의 위치를 저장해두면 이분검색이 필요없어지기 때문에 amortized $O(n)$ 에 해결할 수 있음

4.2 Divide & Conquer Optimization

$O(kn^2) \rightarrow O(kn \log n)$

조건 1) DP 점화식 꼴

$$D[t][i] = \min_{j < i} (D[t - 1][j] + C[j][i])$$

조건 2) $A[t][i]$ 는 $D[t][i]$ 의 답이 되는 최소의 j 라 할 때, 아래의 부등식을 만족해야 함

$$A[t][i] \leq A[t][i + 1]$$

조건 2-1) 비용 C 가 다음의 사각부등식을 만족하는 경우도 조건 2)를 만족하게 됨

$$C[a][c] + C[b][d] \leq C[a][d] + C[b][c] \quad (a \leq b \leq c \leq d)$$

4.3 Knuth Optimization

$O(n^3) \rightarrow O(n^2)$

조건 1) DP 점화식 꼴

$$D[i][j] = \min_{i < k < j} (D[i][k] + D[k][j]) + C[i][j]$$

조건 2) 사각 부등식

$$C[a][c] + C[b][d] \leq C[a][d] + C[b][c] \quad (a \leq b \leq c \leq d)$$

조건 3) 단조성

$$C[b][c] \leq C[a][d] \quad (a \leq b \leq c \leq d)$$

결론) 조건 2, 3을 만족한다면 $A[i][j]$ 를 $D[i][j]$ 의 답이 되는 최소의 k 라 할 때, 아래의 부등식을 만족하게 됨

$$A[i][j - 1] \leq A[i][j] \leq A[i + 1][j]$$

3중 루프를 돌릴 때 위 조건을 이용하면 최종적으로 시간복잡도가 $O(n^2)$ 이 됨

5 Graph

5.1 SCC (Tarjan)

5.2 SCC (Kosaraju)

```

const int MAXN = 100;
vector<int> graph[MAXN], grev[MAXN];
int visit[MAXN], vcnt;
int scc_idx[MAXN], scc_cnt;
vector<int> emit;

```

```

void dfs(int nod, vector<int> graph[]) {
    visit[nod] = vcnt;
    for (int next : graph[nod]) {
        if (visit[next] == vcnt) continue;
        dfs(next, graph);
    }
    emit.push_back(nod);
}

```

```

// find SCCs in given graph
// O(V+E)

```

```

void get_scc() {
    scc_cnt = 0;
    vcnt = 1;
    emit.clear();
    memset(visit, 0, sizeof(visit));

```

```

    for (int i = 0; i < n; i++) {
        if (visit[i] == vcnt) continue;
        dfs(i, graph);
    }

```

```

++vcnt;
for (auto st : vector<int>(emit.rbegin(), emit.rend())) {
    if (visit[st] == vcnt) continue;
    emit.clear();
    dfs(st, grev);
    ++scc_cnt;
    for (auto node : emit)
        scc_idx[node] = scc_cnt;
}
}

```

5.3 2-SAT

$(b_x \vee b_y) \wedge (\neg b_x \vee b_z) \wedge (b_z \vee \neg b_x) \wedge \dots$ 같은 form을 2-CNF라고 함. 주어진 2-CNF 식을 참으로 하는 $\{b_1, b_2, \dots\}$ 가 존재하는지, 존재한다면 그 값은 무엇인지 구하는 문제를 2-SAT 이라 함.

boolean variable b_i 마다 b_i 를 나타내는 정점, $\neg b_i$ 를 나타내는 정점 2개를 만들. 각 clause $b_i \vee b_j$ 마다 $\neg b_i \rightarrow b_j, \neg b_j \rightarrow b_i$ 이렇게 edge를 이어줌. 그렇게 만든 그래프에서 SCC를 다 구함. 어떤 SCC 안에 b_i 와 $\neg b_i$ 가 같이 포함되어있다면 해가 존재하지 않음. 아니라면 해가 존재함.

해가 존재할 때 구체적인 해를 구하는 방법. 위에서 SCC를 구하면서 SCC DAG를 만들어 준다. 거기서 위상정렬을 한 후, 앞에서부터 SCC를 하나씩 봐준다. 현재 보고있는 SCC 에 b_i 가 속해있는데 애가 $\neg b_i$ 보다 먼저 등장했다면 $b_i = \text{false}$, 반대의 경우라면 $b_i = \text{true}$, 이미 값이 assign되었다면 pass.

5.4 BCC, Cut vertex, Bridge

```

const int MAXN = 100;
vector<pair<int, int>> graph[MAXN]; // { next vertex id, edge id }
int up[MAXN], visit[MAXN], vtime;
vector<int> stk;

vector<int> cut_vertex;
vector<int> bridge;
int bcc_idx[MAXN], bcc_cnt;

void dfs(int nod, int par_edge) {
    up[nod] = visit[nod] = ++vtime;
    int child = 0;
    for (const auto& e : graph[nod]) {
        int next = e.first, edge_id = e.second;
        if (edge_id == par_edge) continue;
        if (visit[next] == 0) {
            stk.push_back(next);
            ++child;
            dfs(next, edge_id);
            if (up[next] == visit[next]) bridge.push_back(edge_id);
            if (up[next] >= visit[nod]) {

```

```

                ++bcc_cnt;
                do {
                    bcc_idx[stk.back()] = bcc_cnt;
                    stk.pop_back();
                } while (!stk.empty() && stk.back() != nod);
                bcc_idx[nod] = bcc_cnt;
            }
            up[nod] = min(up[nod], up[next]);
        }
        else
            up[nod] = min(up[nod], visit[next]);
    }
    if ((par_edge != -1 && child >= 1 && up[nod] == visit[nod])
        || (par_edge == -1 && child >= 2))
        cut_vertex.push_back(nod);
}

// find BCCs & cut vertexs & bridges in undirected graph
// O(V+E)
void get_bcc() {
    vtime = 0;
    memset(visit, 0, sizeof(visit));
    cut_vertex.clear();
    bridge.clear();
    memset(bcc_idx, 0, sizeof(bcc_idx));
    bcc_cnt = 0;
    for (int i = 0; i < n; ++i) {
        if (visit[i] == 0)
            dfs(i, -1);
    }
}

```

5.5 Lowest Common Ancestor

```

const int MAXN = 100;
const int MAXLN = 9;
vector<int> tree[MAXN];
int depth[MAXN];
int par[MAXLN][MAXN];

void dfs(int nod, int parent) {
    for (int next : tree[nod]) {
        if (next == parent) continue;
        depth[next] = depth[nod] + 1;
        par[0][next] = parent;
        dfs(next, nod);
    }
}

void prepare_lca() {
    const int root = 0;
    dfs(root, -1);
    par[0][root] = root;
    for (int i = 1; i < MAXLN; ++i)
        for (int j = 0; j < n; ++j)

```

```

        par[i][j] = par[i - 1][par[i - 1][j]];
    }

    // find lowest common ancestor in tree between u & v
    // assumption : must call 'prepare_lca' once before call this
    // O(logV)
    int lca(int u, int v) {
        if (depth[u] < depth[v]) swap(u, v);
        if (depth[u] > depth[v]) {
            for (int i = MAXLN - 1; i >= 0; --i)
                if (depth[u] - (1 << i) >= depth[v])
                    u = par[i][u];
        }
        if (u == v) return u;
        for (int i = MAXLN - 1; i >= 0; --i) {
            if (par[i][u] != par[i][v]) {
                u = par[i][u];
                v = par[i][v];
            }
        }
        return par[0][u];
    }
}

```

5.6 Heavy-Light Decomposition

5.7 Bipartite Matching (Hopcroft-Karp)

```

// in: n, m, graph
// out: match, matched
// vertex cover: (reached[0][left_node] == 0) || (reached[1][right_node] == 1)
// O(E*sqrt(V))
struct BipartiteMatching {
    int n, m;
    vector<vector<int>> graph;
    vector<int> matched, match, edgeview, level;
    vector<int> reached[2];
    BipartiteMatching(int n, int m) : n(n), m(m), graph(n), matched(m, -1),
        match(n, -1) {}

    bool assignLevel() {
        bool reachable = false;
        level.assign(n, -1);
        reached[0].assign(n, 0);
        reached[1].assign(m, 0);
        queue<int> q;
        for (int i = 0; i < n; i++) {
            if (match[i] == -1) {
                level[i] = 0;
                reached[0][i] = 1;
                q.push(i);
            }
        }
        while (!q.empty()) {
            auto cur = q.front(); q.pop();

```

```

            for (auto adj : graph[cur]) {
                reached[1][adj] = 1;
                auto next = matched[adj];
                if (next == -1) {
                    reachable = true;
                }
                else if (level[next] == -1) {
                    level[next] = level[cur] + 1;
                    reached[0][next] = 1;
                    q.push(next);
                }
            }
        }
        return reachable;
    }

    int findpath(int nod) {
        for (int &i = edgeview[nod]; i < graph[nod].size(); i++) {
            int adj = graph[nod][i];
            int next = matched[adj];
            if (next >= 0 && level[next] != level[nod] + 1) continue;
            if (next == -1 || findpath(next)) {
                match[nod] = adj;
                matched[adj] = nod;
                return 1;
            }
        }
        return 0;
    }

    int solve() {
        int ans = 0;
        while (assignLevel()) {
            edgeview.assign(n, 0);
            for (int i = 0; i < n; i++)
                if (match[i] == -1)
                    ans += findpath(i);
        }
        return ans;
    }
};

```

5.8 Maximum Flow (Dinic)

```

// usage:
// MaxFlowDinic::init(n);
// MaxFlowDinic::add_edge(0, 1, 100, 100); // for bidirectional edge
// MaxFlowDinic::add_edge(1, 2, 100); // directional edge
// result = MaxFlowDinic::solve(0, 2); // source -> sink
// graph[i][edgeIndex].res -> residual
//
// in order to find out the minimum cut, use 'l'.
// if l[i] == 0, i is unreachable.
//
// O(V*V*E)

```



```
// with unit capacities, O(min(V^(2/3), E^(1/2)) * E)
struct MaxFlowDinic {
    typedef int flow_t;
    struct Edge {
        int next;
        int inv; /* inverse edge index */
        flow_t res; /* residual */
    };
    int n;
    vector<vector<Edge>> graph;
    vector<int> q, l, start;

    void init(int _n) {
        n = _n;
        graph.resize(n);
        for (int i = 0; i < n; i++) graph[i].clear();
    }
    void add_edge(int s, int e, flow_t cap, flow_t caprev = 0) {
        Edge forward{ e, graph[e].size(), cap };
        Edge reverse{ s, graph[s].size(), caprev };
        graph[s].push_back(forward);
        graph[e].push_back(reverse);
    }
    bool assign_level(int source, int sink) {
        int t = 0;
        memset(&l[0], 0, sizeof(l[0]) * l.size());
        l[source] = 1;
        q[t++] = source;
        for (int h = 0; h < t && !l[sink]; h++) {
            int cur = q[h];
            for (const auto& e : graph[cur]) {
                if (l[e.next] || e.res == 0) continue;
                l[e.next] = l[cur] + 1;
                q[t++] = e.next;
            }
        }
        return l[sink] != 0;
    }
    flow_t block_flow(int cur, int sink, flow_t current) {
        if (cur == sink) return current;
        for (int& i = start[cur]; i < graph[cur].size(); i++) {
            auto& e = graph[cur][i];
            if (e.res == 0 || l[e.next] != l[cur] + 1) continue;
            if (flow_t res = block_flow(e.next, sink, min(e.res, current))) {
                e.res -= res;
                graph[e.next][e.inv].res += res;
                return res;
            }
        }
        return 0;
    }
    flow_t solve(int source, int sink) {
        q.resize(n);
        l.resize(n);
        start.resize(n);

```

```
        flow_t ans = 0;
        while (assign_level(source, sink)) {
            memset(&start[0], 0, sizeof(start[0]) * n);
            while (flow_t flow = block_flow(source, sink, numeric_limits<
                flow_t>::max()))
                ans += flow;
        }
        return ans;
    }
};
```

5.9 Min-cost Maximum Flow

6 Geometry

6.1 Basic Operations

6.2 Compare angles

6.3 Convex Hull

```
// find convex hull
// O(n*logn)
vector<Point> convex_hull(vector<Point>& dat) {
    if (dat.size() <= 3) return dat;
    vector<Point> upper, lower;
    sort(dat.begin(), dat.end(), [](const Point& a, const Point& b) {
        return (a.x == b.x) ? a.y < b.y : a.x < b.x;
    });
    for (const auto& p : dat) {
        while (upper.size() >= 2 && ccw(++upper.rbegin(), *upper.rbegin(), p)
            >= 0) upper.pop_back();
        while (lower.size() >= 2 && ccw(++lower.rbegin(), *lower.rbegin(), p)
            <= 0) lower.pop_back();
        upper.emplace_back(p);
        lower.emplace_back(p);
    }
    upper.insert(upper.end(), ++lower.rbegin(), --lower.rend());
    return upper;
}
```

6.4 Polygon Cut

6.5 Pick's theorem

격자점으로 구성된 simple polygon이 주어짐. i 는 polygon 내부의 격자점 수, b 는 polygon 선분 위 격자점 수, A 는 polygon의 넓이라고 할 때, 다음과 같은 식이 성립한다.

$$A = i + \frac{b}{2} - 1$$

7 String

7.1 KMP

```
typedef vector<int> seq_t;

void calculate_pi(vector<int>& pi, const seq_t& str) {
    pi[0] = -1;
    int j = -1;
    for (int i = 1; i < str.size(); i++) {
        while (j >= 0 && str[i] != str[j + 1]) j = pi[j];
        if (str[i] == str[j + 1])
            pi[i] = ++j;
        else
            pi[i] = -1;
    }
}

// returns all positions matched
// O(|text|+|pattern|)
vector<int> kmp(seq_t& text, seq_t& pattern) {
    vector<int> pi(pattern.size());
    vector<int> ans;
    if (pattern.size() == 0) return ans;
    calculate_pi(pi, pattern);
    int j = -1;
    for (int i = 0; i < text.size(); i++) {
        while (j >= 0 && text[i] != pattern[j + 1]) j = pi[j];
        if (text[i] == pattern[j + 1]) {
            j++;
            if (j + 1 == pattern.size()) {
                ans.push_back(i - j);
                j = pi[j];
            }
        }
    }
    return ans;
}
```

7.2 Aho-Corasick

```
#include <algorithm>
#include <vector>
#include <queue>
using namespace std;

struct AhoCorasick
{
    const int alphabet;
    struct node {
        node() {}
        explicit node(int alphabet) : next(alphabet) {}
        vector<int> next, report;
```

```
        int back = 0, output_link = 0;
    };
    int maxid = 0;
    vector<node> dfa;
    explicit AhoCorasick(int alphabet) : alphabet(alphabet), dfa(1, node(
        alphabet)) {}
    template<typename InIt, typename Fn> void add(int id, InIt first, InIt
        last, Fn func) {
        int cur = 0;
        for (; first != last; ++first) {
            auto s = func(*first);
            if (auto next = dfa[cur].next[s]) cur = next;
            else {
                cur = dfa[cur].next[s] = (int)dfa.size();
                dfa.emplace_back(alphabet);
            }
        }
        dfa[cur].report.push_back(id);
        maxid = max(maxid, id);
    }
    void build() {
        queue<int> q;
        vector<char> visit(dfa.size());
        visit[0] = 1;
        q.push(0);
        while(!q.empty()) {
            auto cur = q.front(); q.pop();
            dfa[cur].output_link = dfa[cur].back;
            if (dfa[dfa[cur].back].report.empty())
                dfa[cur].output_link = dfa[dfa[cur].back].output_link;
            for (int s = 0; s < alphabet; s++) {
                auto &next = dfa[cur].next[s];
                if (next == 0) next = dfa[dfa[cur].back].next[s];
                if (visit[next]) continue;
                if (cur) dfa[next].back = dfa[dfa[cur].back].next[s];
                visit[next] = 1;
                q.push(next);
            }
        }
    }
    template<typename InIt, typename Fn> vector<int> countMatch(InIt first,
        InIt last, Fn func) {
        int cur = 0;
        vector<int> ret(maxid+1);
        for (; first != last; ++first) {
            cur = dfa[cur].next[func(*first)];
            for (int p = cur; p; p = dfa[p].output_link)
                for (auto id : dfa[p].report) ret[id]++;
        }
        return ret;
    }
};
```

7.3 Suffix Array with LCP

7.4 Suffix Tree

7.5 Manacher's Algorithm

```
// find longest palindromic span for each element in str
// O(|str|)
void manacher(const string& str, int plen[]) {
    int r = -1, p = -1;
    for (int i = 0; i < str.length(); ++i) {
        if (i <= r)
            plen[i] = min((2 * p - i >= 0) ? plen[2 * p - i] : 0, r - i);
        else
            plen[i] = 0;
        while (i - plen[i] - 1 >= 0 && i + plen[i] + 1 < str.length()
            && str[i - plen[i] - 1] == str[i + plen[i] + 1]) {
            plen[i] += 1;
        }
        if (i + plen[i] > r) {
            r = i + plen[i];
            p = i;
        }
    }
}
```

8 Miscellaneous

8.1 Fast I/O

```
namespace fio {
    const int BSIZE = 524288;
    char buffer[BSIZE];
    int p = BSIZE;
    inline char readChar() {
        if(p == BSIZE) {
            fread(buffer, 1, BSIZE, stdin);
            p = 0;
        }
        return buffer[p++];
    }
    int readInt() {
        char c = readChar();
        while ((c < '0' || c > '9') && c != '-') {
            c = readChar();
        }
        int ret = 0; bool neg = c == '-';
        if (neg) c = readChar();
        while (c >= '0' && c <= '9') {
            ret = ret * 10 + c - '0';
            c = readChar();
        }
    }
}
```

```
    }
    return neg ? -ret : ret;
}
```

8.2 Magic Numbers